Context-Sensitive Staged Static Taint Analysis for C using LLVM

Xavier NOUMBISSI NOUNDOU
{xavier.noumbis@gmail.com}

Abstract

Software security vulnerabilities are a major threat for software systems. In the worst case, vulnerabilities in software enable users to gain unauthorized access or unauthorized control of an application. A large amount of software security vulnerability exploits such as buffer overflows, SQL injections, cross-site scripting attacks, etc. are caused by data flowing from untrusted program input sources into sensible program functions. We define a *tainted path* as a program execution path from an untrusted program input source into a sensible program location. This paper presents a static taint analysis that computes tainted paths in C programs and that doesn't require any program annotations. Our static taint analysis algorithm is built upon the iterative dataflow framework [11, 17] and has been implemented in the tool SAINT (Simple Static Taint Analysis Tool). Our static taint analysis is interprocedural, flow-sensitive, and developers can choose to run it either with context-sensitivity or without. We have implemented our analysis using the LLVM compiler infrastructure.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms Software Security Vulnerabilities, Program Analysis, Static Code Analysis, Static Taint Analysis

Keywords tainted path, static taint analysis, static code analysis

1. Introduction

Software vulnerabilities are security threats that exist in an application. Software vulnerabilities allow malovelent users to exercise unauthorized control of the application through supplied input. There are several kinds of software vulnerabilities: buffer overflows, format string attacks, SQL injection, etc. Researchers have worked on dynamic [6, 10], static [2, 4, 5, 9, 10, 15, 22, 25], and hybrid techniques [24] to find security vulnerabilities in software.

This paper introduces the concept of *tainted path*. A tainted path is a program execution path from a program input source into a sensible program location. A tainted path represent a software vulnerability. This paper presents a static taint analysis that computes tainted data and tainted paths in C programs. Our implementation of the taint analysis uses the LLVM framework [12], and does not require user annotations. Our static taint analysis is flow-sensitive, interprocedural, and developers can choose to run it with context-sensitivity or without. In taint analysis, a *source* is a program location that allows a value from the environment into the program. This may occur through the return value of a system call, user input, etc. A value from the program environment that has not been *validated* and *sanitized* is called a *tainted value*. A *sink* is a program location that uses a tainted value .

Data validation is the process of checking that data has the expected form. For instance, checking that a string input has the format of an email address. *Data sanitization* is the process of checking that validated data is safe in a particular context. For instance, escaping string input before using it in a SQL query. A function that sanitizes application's external input is called a *sanitizer*. Once a value has been sanitized, it is tagged as *not tainted*. In the following, we assume that sanitized data has been validated.

Static taint analysis searches for tainted values and warn developers for each tainted value so they can validate and sanitize the tainted value to avoid software vulnerability exploits at runtime. Taint analysis proceeds by first tagging values from sources as tainted. Once tagged, the tainted values are propagated through the entire program.

Taint propagation is the process of marking values as *tainted* if they result from an operation that involved tainted data. This can be an arithmetic operation (addition, multiplication, etc.), a program assignment or other type of program instructions. Finally, a taint analysis emits a warning whenever a tainted value is used at a sink location. Taint propagation can be *data-flow* or *control-flow* based. Data-flow based taint propagation exists due to data dependencies in the program (e.g. assigning the value of tainted variable s_u to another variable s_d). Control-flow based taint propagation is due to control dependencies (e.g. if tainted variable s_t is used in a branch condition, values from program instructions inside that branch become tainted.). Data-flow based taint propagation is also called *explicit taint propagation*, and control-flow based taint propagation is called *implicit taint propagation*. Our static taint analysis searches for *tainted paths* and implements both control-flow and data-flow based taint propagation.

This paper makes the following contributions:

- It introduces the concept of *tainted path*, which is a program execution path from a taint source to a taint sink.
- It shows that several static analysis problems can be reduced to a tainted path computation problem.
- It describes SAINT, a whole-program static taint analysis that is flow-sensitive, interprocedural and context-sensitive. SAINT computes tainted paths in C programs and is available for download at `https://github.com/xaviernoumbis/saint`

ABSTRACT INTRUCTIONS	DESCRIPTION	CODE IN C	FORMAL DESCRIPTION
ALLOC	Memory allocation	$v = malloc(...)$	$v \in \mathcal{T}$
COPY	Copy instruction	$p = q$	$p, q \in \mathcal{T}$
LOAD	Load instruction	$p = *q$	$p \in \mathcal{T}, q \in \mathcal{A}$
STORE	Store instruction	$*p = q$	$p \in \mathcal{A}, q \in \mathcal{T}$
CALL	Call instruction	$r = call$ func (p)	$r \in \mathcal{T}$

Table 1: LLVM abstract instructions types. In LLVM intermediate representation, $\mathcal{A}$ and $\mathcal{T}$ represent respectively the set of address-taken variables and the set of top-level variables.

SAINT's taint analysis is sound (i.e. all tainted paths are reported).

To the best of our knowledge, SAINT is the only tool that implements static taint analysis for a static language using the iterative dataflow framework [11, 17]. Please look at Table 7 in Section 6.

The paper is organized as follows: Section 2 introduces the concept of tainted path. Section 3 introduces our running example, and Section 4 gives an overview of the LLVM intermediate representation. Section 5 presents the taint analysis algorithm, and Section 6 presents experimental results. Finally, Section 7 discusses related work and Section 8 concludes.

2. Tainted Paths

```
1       void mysql_taint(uint); //taint sink

3       uint calculate(uint x) {
4   L1:   uint sum = 0;
5   L2:   uint i = 0;
6   L3:   if (x == 2)
7   L4:     scanf("%d", &sum); //taint source
8   L5:   else
9   L6:     sum = 0;
10  L7:   if (i >= x)
11  L8:     goto L12;
12  L9:   sum = sum + i;
13  L10:  i = i + 1;
14  L11:  mysql_taint(i);
15  L12:  goto L6;
16  L13:  return sum;
17      }
```

Figure 1: Code example in three-address format

In this section we introduce the term *tainted path*. We define a *tainted path* as a program execution path from a taint source to a taint sink. Let us consider the three-address code in Figure 1, we represent a line of code with Lx where L means line and x represents a line number. The program path $< \text{L4}, \text{L7}, \text{L9}, \text{L10}, \text{L11} >$ defines a tainted path while program paths $< \text{L4}, \text{L7}, \text{L9}, \text{L10} >$ and $< \text{L4}, \text{L7}, \text{L8} >$ don't define tainted paths. We postulate that several static analysis problems can be reduced to a combination of tainted paths computation and test data generation. For instance, this is the case for the following static analysis problems:

- Buffer-overflow detection.
- SQL injection vulnerability detection.
- Format string vulnerability detection.
- Automatic test cases and test data generation.

2.1 Buffer-overflow detection

2.2 Automatic test cases and test data generation

Given a tainted path, one can combine *symbolic execution* and *constraint solving* to generate data needed to execute the tainted pah. The generated data, along with the given tainted path represent a test case (or an exploit). Developers can then change the code that cause the bug (or vulnerability).

3. Motivating Example

```
1   void func_sql(int); //sink

3   int compute(int x) {
4     int sum = -1;
5     if (x == 2)
6       sum = func_sql (x);
7     return sum;
8   }

10  int main() {
11    int x, y;
12    scanf("%d", &x);
13    y = compute(x);
14    return 0;
15  }
```

Figure 2: Motivating example

This paper uses an example inspired from the example described in [3]. Figure 2 shows 2 functions: `main` and `compute`. In `main`, the function `scanf` from the C standard input/ouput library gets an integer input from the user at line 3 and stores it in variable `x`. `x` then becomes tainted bacause it holds a value from the environment which has not been validated and sanitized. `x` is later used as argument to function `compute` at line 4. In `compute`, variable `sum` gets tainted at line 10 through function `scanf`. The formal parameter `x` gets tainted if only if a tainted actual argument was passed at calling sites. This is for instance the case at line 4 in function `main`. Observe that if a tainted parameter `x` is used at the calling site of line 4 in function `main`: this leads to a case of control-flow based taint propagation at line 10 and at line 11 of function `compute`. Variable `sum` also becomes tainted at line 11 because the statement at line 11 is control-dependent of the conditional expression at line 10.

4. LLVM

This section gives an overview of LLVM[1] (*Low Level Virtual Machine*) and its intermediate representation (IR), which we use as

[1] http://llvm.org

ABSTRACT INSTRUCTIONS	CODE IN C	GEN-SET	KILL-SET
ALLOC	s: $v = malloc(...)$	$\emptyset$	$\emptyset$
COPY	s: $p = q$	$\{p\}$ iff $q \in \text{IN}[s]$	$\emptyset$
LOAD	s: $p = *q$	$\{t_j \mid t_j = toplevel(a_j) \wedge a_j \in points\_to_{[s]}(q) \wedge t_j \in \text{IN}[s]\}$	$\emptyset$
STORE	s: $*p = q$	$\{t_j \mid t_j = toplevel(a_j) \wedge a_j \in points\_to_{[s]}(p)\}$ iff $q \in \text{IN}[s]$	$\emptyset$

Table 2: Gen- and kill-sets for the abstract instructions **ALLOC**, **COPY**, **LOAD**, and **STORE**

basis for the description of our taint analysis. LLVM is a compiler framework [12] that contains several components and libraries that help developers in building compilers and compiler tools (e.g. static analyses, etc.). LLVM primarily processes source code written in C, C++, and Objective C. LLVM libraries are written in C++. Table 1 shows the abstract instruction types we consider in the LLVM IR for our analysis. In the following, we present the LLVM intermediate representation. Our presentation is based on the descriptions given by Hardekopf et al. [8] and by Lhoták et al. [14]. LLVM's IR uses partial static single assignment (partial SSA) and assumes the existence of two types of variables in C code: *top-level* variables and *address-taken* variables.

4.1 Top-level variables

Top-level variables are variables that are never accessed via a pointer in the program code. LLVM converts top-level variables into SSA form when building the LLVM IR. The memory address of top-level variables is never copied to another variable (i.e. they are never applied the address-of operator (&) in the C programming language). In the LLVM IR, top-level variables are only accessed using **ALLOC** and **COPY** instructions. This paper denotes the set of top-level variables with $\mathcal{T}$. In Figure 2 $b1$, and $b2$ are top-level variables ($\{b1, b2\} \in \mathcal{T}$).

4.2 Address-taken variables

Address-taken variables are never accessed directly through their first declared name. Address-taken variables are only accessed indirectly with pointer variables and **LOAD** and **STORE** instructions. In fact, address-taken variables are those ones on which the address-of operator (&) was applied. This paper uses $\mathcal{A}$ for the set of all address-taken variables. Variable x in Figure 2 is for instance an address-taken variable ($x \in \mathcal{A}$).

4.3 Representation of program expressions

5. Staged Static Taint Analysis

Our taint analysis is interprocedural and runs either context-insensitively or context-sensitively. Any form of the interprocedural analysis is always preceded by an intraprocedural analysis that computes initial taint information that is reused by the interprocedural analyses. The intraprocedural analysis detects taint sources and initializes a summary table which contains taint information about program functions' formal parameters and return value. The use of a summary table allows fast access to key information about program procedures. This is especially useful during the subsequent interprocedural phases. For instance, the intraprocedural analysis would detect that variable `sum` of procedure `compute` in Figure 2, which also holds the return value of `compute`, may be tainted due to the call to `scanf` at line 12. Figure 3 shows the architecture of our taint analysis SAINT and Table 2 shows the transfer functions for the abstract program statements **ALLOC**, **COPY**, **LOAD**, **STORE**, and **CALL**. These transfer functions apply for the intraprocedural and the interprocedural analyses.

5.1 Taint sources and taint sinks

Program statements that initially taint variables (*taint sources*) are discovered during the intraprocedural analysis, described later in this section. The analysis handles per default a subset of the C standard library as taint sources: `getc, scanf, gets, fopen`, etc. Functions that use tainted variables (*taint sinks*) are gradually discovered during the various phases of the analysis. SAINT has a configuration file where developers can register additional taint source and taint sink functions.

5.2 Taint propagation

SAINT performs *explicit* and *implicit* taint propagation (data- and control-flow taint propagation). Explicit taint propagation tracks variables that are tainted due to assignment statements. The assignment in line 6 of Figure 2 is an instance of explicit taint propagation. Variable y becomes tainted since it gets assigned the return value of `compute`, which is a tainted value.
Implicit taint propagation takes into account tainted variables used in control conditions. In Figure 2 for instance, variable x is passed to the function `compute` as actual parameter as a tainted variable during the context-sensitive analysis. This implies that variables sum and i becomes tainted at line 15 of Figure 2 because x is tainted and is part of the for-loop boolean condition at line 14.

5.3 Sanitizers

Sanitizers are functions that developers use to make sure that tainted data are safe to use in sensible (or vulnerable) program functions. SAINT creates *kill-sets* whenever a sanitizer is found on a tainted path.

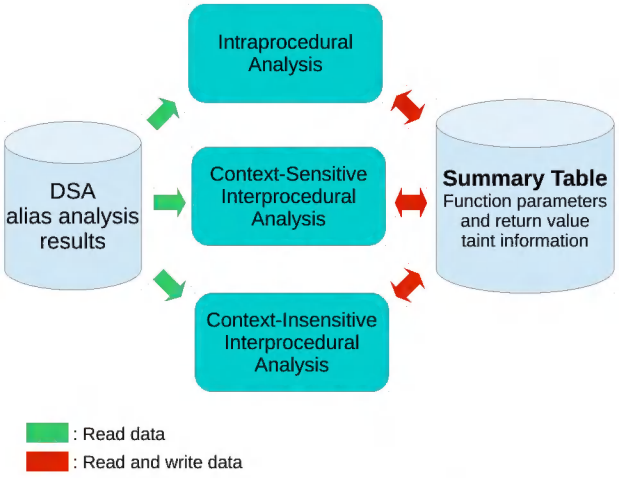

Figure 3: SAINT analysis architecture

5.4 Formalisms

This paper uses the following elements to describe the taint analysis as an instance of the iterative dataflow analysis framework [17]:

Var is the set of all program variables, $Proc$ is the set of all program functions and procedures[2]. $Inst$ is the set of all program statements, $formals : Proc \rightarrow 2^{Var}$ returns the set of formal parameters of a function, $toplevel : \mathcal{A} \rightarrow \mathcal{T}$ returns the top level variable of an address-taken variable, $taint : Proc \times Integer \rightarrow bool$ returns `true` if function f always taints its k^{th} formal parameter, and $aliases : Var \rightarrow 2^{Var}$ returns the alias set of a variable as returned by the pointer analysis.

The analysis dataflow set is $Inst \times 2^{Var}$. 2^{Var}, the powerset of all program variables (Var) is by definition a lattice. In effect, $Inst \times 2^{Var}$ is a mapping from the set of all program instructions $Inst$ into 2^{Var}. $Inst \times 2^{Var}$ is therefore a lattice by definition[3].

At a statement labelled s, the incoming dataflow set $IN[s]$ is the set of all program variables that are tainted before statement s. If a variable v is not tainted before statement s, then $v \notin IN[s]$; otherwise $IN[s]$ contains v.

The execution of the transfer function at a statement labelled s eventually discover new tainted variables. $OUT[s]$ describes the new set of tainted variables after statement s.

The function $sumTable : Proc \times Integer \rightarrow \{True, False\}$ describes the summary table, and reveals for a procedure $f \in Proc$ and its k^{th} formal parameter whether the k^{th} formal parameter is tainted or not. For instance $sumTable[f][2] = True$ means that the third formal parameter of function f is tainted.

We use ret to represent the return value of a function f. ret is tainted implies that $sumTable[f][ret] = True$, otherwise $sumTable[f][ret] = False$. $points\_to_{[\bar{s}]}(q)$ describes the set of aliases of variable q before statement s.
$read\_taint\_arg : Proc \rightarrow Integer$ reads

5.5 Handling of pointers

Our taint analysis is designed to work using the results of a previously computed pointer analysis. SAINT uses the pointer analysis `DSA (Data Structure Analysis)` [13] which is implemented in the tool `poolalloc`[4]. `DSA` is a field- and context-sensitive pointer analysis. `DSA` uses full heap cloning (by acyclic call paths) and scales well with programs in a size range of 100K-200K lines of C code. $points\_to_{[\bar{s}]}(q)$ describes the set of aliases of variable q before statement s. Our taint analysis uses the results of a pointer analysis via the function $points\_to$ to update dataflow sets.

5.6 Intraprocedural analysis

The intraprocedural analysis always runs first, before any interprocedural analysis, and is responsible for discovering taint sources. During the intraprocedural analysis, program functions are analyzed in the reverse topological order of the call graph (i.e starting from the leaves of the callgraph). The analysis works on each function body and do not take into account interprocedural control flow. The computed data flow sets are reused later by the subsequent interprocedural analyses. In particular, taint information about function formal parameters and return value is kept in a summary table. All variables tainted due to source functions are found during the intraprocedural analysis. In Figure 2 for instance, the intraprocedural analysis detects that variable `sum` in function `compute` may be tainted at line 12 due to the call to `scanf`, which the analysis considers as a taint source. Algorithm 1 shows the flow function for function call statements, which handles the discovery of taint sources during the intraprocedural analysis. Flow functions for all other statement types are same as the ones in Algorithm 3 of Appendix A. Table 4 illustrates the summary table after execution of

input : caller : $Proc$, s : $Inst$, k : $Integer$
output:
switch `TypeOf`(s) **do**
  case `CALL` [r = *call* `source`($a_0, a_1, ..., a_n$)]
    k := `read_taint_arg`(`source`)
    foreach $v_j \in points\_to_{[\bar{s}]}(a_k)$ **do**
      if $v_j \in$ IN[s] **then**
        OUT[s] := OUT[s] $\cup \{v_j\}$
      end
    end
  end
endsw
foreach $f_k \in$ `formals`(caller) **do**
  if IN[s] $\neq$ OUT[s] **and** $f_k \in$ OUT[s] **then**
    sumTable[caller][k] := *True*
  end
end

Algorithm 1: Intraprocedural analysis transfer function for `CALL` statements.

the intraprocedural analysis of the running example presented in Section 3.

	Untainted variables	Tainted variables
main	$\{b_1, b_2, y, ret\}$	$\{x\}$
compute	$\{x, i\}$	$\{sum\}$

Table 4: Summary table after the intraprocedural analysis. ret represents the return value of the function.

input : caller : $Proc$, s : $Inst$
output:
switch `TypeOf`(s) **do**
  case `CALL` [r = *call* `func`($a_0, a_1, ..., a_n$)]
    if *True* = sumTable[`func`][ret] **then**
      foreach $v_j \in points\_to_{[\bar{s}]}$(r) **do**
        if $v_j \in$ IN[s] **then**
          OUT[s] := OUT[s] $\cup \{v_j\}$
        end
      end
    end
    foreach k $\in \{0, 1, ..., n\}$ **do**
      if *True* = sumTable[`func`][k] **then**
        foreach $v_j \in points\_to_{[\bar{s}]}(a_k)$ **do**
          if $v_j \in$ IN[s] **then**
            OUT[s] := OUT[s] $\cup \{v_j\}$
          end
        end
      end
    end
  end
endsw

Algorithm 2: Context-insentive interprocedural transfer function for `CALL` statements

5.7 Context-insensitive analysis

The context-insensitive analysis algorithm performs in the topological order of the call graph (i.e. the algorithm runs from the program entry point to the leaves of the call graph). The context-insensitive

[2] We will use the terms function and procedure interchangeably in the remainder of this paper

[3] Please consult the axiomatic propositions that define lattices

[4] https://github.com/poolalloc

PROGRAM (VERSION)	DESCRIPTION	#SLOC	TAINTED VALUES (%)	#TAINTED PATHS	TIME (SECONDS)
`mongoose` (4.1)	Web server	$4k$			$2.14s$
`vlc-input` (2.1.2)	Media player	$16k$			$0.03s$
`openssl-ssl` (1.0.1f)	Communication Protocol	$40k$			$0.44s$
`apache` (2.4.7)	Web server	$144k$			n/a

Table 3: SAINT's taint analysis experiment results

	Untainted variables	Tainted variables
main	$\{b_1, b_2, ret\}$	$\{x, y\}$
compute	$\{x, i\}$	$\{sum\}$

Table 5: Summary table after the context-insensitive analysis

analysis only uses taint assumptions from the summary table to update data flow sets at program points. For instance, the intraprocedural analysis of function `main` marks the return value of `compute` (stored in variable `sum`) as tainted in the summary table. At line 6 in `main`, the context-insensitive analysis would take into account that the return value of `compute` is stored into variable `y`. Thus, `y` becomes a tainted variable during the context-insensitive analysis. Algorithm 2 in the following illustrates the context-insensitive flow function for `CALL` statements.

The context-insensitive algorithm implements the interprocedural *functional approach* by Sharir and Pnueli [23]. This is evident because SAINT only uses the interprocedural information stored in the summary table during the context-insenstive analysis.

Table 5 illustrates the summary table after execution of the context-insensitive analysis of the running example presented in Section 3.

	Untainted variables	Tainted variables
main	$\{ret\}$	$\{x, y\}$
compute	$\{i\}$	$\{x, sum, ret\}$

Table 6: Summary table after the context-sensitive analysis. ret represents the return value of the function.

5.8 Context-sensitive analysis

The context-sensitive analysis algorithm works in the topological order of the call graph, and uses information from the summary table that were produced by previous analyses. Even if the context-insensitive analysis was run before, the context-sensitive analysis eventually writes more precise information in the summary table. At call sites, the context-sensitive analysis propagates actual parameters taint information from the caller into the callee. At callee exits, newly computed taint information are propagated back from the callee context to the caller context. Algorithm **??** illustrates the flow function for `CALL` statements during the context-sensitive analysis. On the other hand, algorithm 3 in Appendix A presents the full context-sensitive algorithm for all handled type of program statements.

The context-sensitive algorithm implements the interprocedural *call-string approach* by Sharir and Pnueli [23]. The call-string length in SAINT implementation is 2, which means that 2 is the depth at which context-sensitive calls are made. It also means that SAINT only analyzes the first two calling contexts of a recursive procedure. Developers can specify longer or shorter calling contexts. We have implemented the call-string approach in SAINT using recursion, as illustrated in lines $10 - 12$ of Algorithm **??**. Table 6 illustrates the summary table after execution of the context-sensitive analysis of the running example presented in Section 3.

5.9 Arrays and C structures

6. Experimental Evaluation

We evaluated our taint analysis by attempting to detect format string vulnerabilities of `printf`-like functions in four different open-source C programs (`mongoose, vlc, claws, apache`). In the C programming language, *format string vulnerabilities* happen a call to a variable-length function has an incorrect number of arguments. For instance, a call such as

```
printf(buf); // format string vulnerability
```

may cause the program to crash because the function `printf` expects its first parameter to be a format string. For instance `printf("%s", buf)` would be a correct call. For this experiment, the taint analysis emits a warning whenever a format string vulnerability is encountered, or whenever a tainted variable is used in a sink function (e.g. `snprintf`). We ran the analysis on an `Intel i7 @ 3.4GHz` with 4 cores and 16 GB of RAM, running `Debian Linux`. Table 3 shows the analysis results of the taint analysis, ran in the following order: intraprocedural analysis, context-sensitive analysis, and context-insensitive analysis. The first column of the table entails the program name and its version in parenthesis.

We could not get the taint analysis running on the `apache` web server because of a crash happening during initialization of the `DSA` points-to analysis. We note that the analysis of `vlc-input` (the input module of `vlc`) with only $16k$ LOC requires more analysis time than `claws` which has $144k$ LOC. We believe this is due to the high amount of tainted values within `vlc-input`. Because of the high amount of tainted values, the analysis spends more time when checking if a value (argument of call) is tainted or not. We believe that our current implementation performs in an acceptable time, since the developer has to wait for a maximum of 56 seconds for a project with around $140k$ lines of code.

7. Related Work

There has been a lot of work in the area of taint analysis and its applications. This section presents some taint analysis-based projects that rely on static, dynamic, hybrid analysis or symbolic evaluation.

7.1 Static analysis for vulnerability detection

Parfait from Oracle Labs checks for bugs in C programs [5]. `Parfait` is built on top of LLVM and uses a demand driven analysis to mitigate scalability issues inherent to standard forward dataflow analysis techniques. `Parfait` does not require annotations from developers and is advertised to scale to millions of lines of code. For security vulnerabilities, `Parfait` implements a taint analysis [21] as pre-processing filter that is linear in the number of statements and dependencies. `Parfait`'s taint analysis is formulated as a graph reachability problem. `Parfait` implements both a context-insensitive and a context-sensitive solution for its taint-analysis, and adds a may-function alias analysis to LLVM to better support the accuracy of the taint analysis. Similarly to `Parfait`, SAINT may also perform either a context-insensitive or a context-sensitive analysis. SAINT is also based on LLVM, and do

```
input  : caller : Proc, s : Inst, k : {1, 2, ..., n}, cnfMax : {1, 2, ..., n}
output:
switch TypeOf(s) do
    case COPY [p = q]
        if q ∈ IN[s] then
            OUT[s] := OUT[s] ∪ {p}
        end
    end
    case LOAD [p = *q]
        foreach a_j ∈ points_to_[s](q) do
            t_j := toplevel(a_j)
            if t_j ∈ IN[s] then
                OUT[s] := OUT[s] ∪ {t_j}
            end
        end
    end
    case STORE [*p = q]
        if q ∈ IN[s] then
            foreach a_j ∈ points_to_[s](p) do
                t_j := toplevel(a_j)
                if t_j ∈ IN[s] then
                    OUT[s] := OUT[s] ∪ {t_j}
                end
            end
        end
    end
    case CALL [r = call func(a_0, a_1, ..., a_n)]
        if caller ≠ func then
            foreach f_j ∈ formals(func) do
                if False = summary[func][j] then
                    foreach v_j ∈ points_to_[s](a_j) do
                        if v_j ∈ IN[s] then
                            OUT[s] := OUT[s] ∪ {v_j}
                        end
                    end
                end
            end
            if k < cnfMax then
                k := k + 1
                csInterFlow(caller, func, k, cnfMax)
            end
            foreach f_j ∈ formals(func) do
                if False = summary[func][j] and OUT[f_j] ≠ IN[f_j] then
                    summary[func][j] := True
                    foreach v_j ∈ points_to_[s](a_j) do
                        if v_j ∈ IN[s] then
                            OUT[s] := OUT[s] ∪ {v_j}
                        end
                    end
                end
            end
        end
    end
endsw
```

Algorithm 3: csInterFlow: Context-sentive analysis algorithm

TOOLS	LANGUAGE	REQUIRES ANNOTATION	TECHNIQUE	AVAILABILITY	APPEARANCE YEAR
PARFAIT [20]	C, C++		Graph reachability algorithm	Commercial	2008
COVERITY	C, C++, Java		?	Commercial	
FORTIFY	C, C++, Java		?	Commercial	
VERACODE	C, C++, Java		?	Commercial	
TAJ [25]	Java		Program slicing	Commercial	2009
PIXY [9]	PHP		Iterative dataflow framework	Research	2006
FLOWDROID [1]	Java (Android)		IFDS framework	Research	2014
CQUAL [22]	C	✓	Type system	Research	2001
STAC [3]	C	✓	Type system	Research	2009
* SAINT	C		Iterative dataflow framework	Research	2015

Table 7: Static taint analysis tools for security vulnerability search

not require any annotations from developers. It is not possible to use a third party alias analysis with `Parfait`. In contrast, SAINT users have the possibility to change the pointer analysis library it uses.

Pixy is a tool that statically scans for cross-site scripting vulnerabilities in PHP scripts [9]. `Pixy` implements a flow-sensitive, context-sensitive dataflow analysis. `Pixy` also creates and uses an alias and literal analysis for PHP.

Livshits et al. present a tool for finding security vulnerabilities in web applications written in Java [15]. The tool is based on `bddbddb` [5], which automatically generates context-sensitive program analyses for specifications written in `Datalog` [27]. `bddbddb` uses binary decisions diagrams to represent and manipulate points-to analysis results for different contexts in a Java program. The authors implement taint propagation on top of the points-to analysis results generated by `bddbddb`. In effect, developers specify vulnerabilities in the `PQL` language [16], which is a syntactic sugar for `Datalog`. That is, each vulnerability specification corresponds to a set of `PQL` queries.

CQual by Shankar et al. detects format string vulnerabilities in C programs [22]. `CQual`'s analysis is based on type qualifiers [7] and type inference. Given a C program, an initial subset of the program is annotated with the type qualifiers `tainted` and `untainted`. `CQual` then uses a set of inference rules to generate type (qualifier) constraints over the program. Pointer reasoning is done via some rules of the type system. The analysis then warns the user of a format string vulnerability whenever the program does not type checks due to an unsatisfiable constraint. According to results published in [22], `CQual` performs on average 85s for a code base of 20k lines of code, and takes 268s to analyze the largest program of 43k. In contrast to `CQual`, users of SAINT may experiment with different pointer analysis libraries and choose among them. SAINT also do not require any annotations.

FLOWDROID [1]

TAJ [25]

7.2 Dynamic analysis for vulnerability detection

TaintDroid

Dytan is a framework that generates dynamic taint analyzes for x86 binaries [6]. `Dytan` does not need access or recompilation of program source code. Developers specify an analysis by giving taint sources, taint sinks, and a propagation policy. Taint sources can be variable names, function-return values, and data read from I/O stream (e.g. file, network connection, etc.). Taint sinks are specified by memory or code location. Sinks can also be based on usage scenarios (e.g. perform a check before execution of `jump` instructions). Clause et al. report performance overhead op up to 30 times for data-flow propagation only. The slowdown goes up to 50 times for combined control- and data-flow propagation. In contrast, SAINT is used during the development phase and does not incur any performance slowdown during program execution.

TaintCheck by Newsome et al. implements a dynamic taint analysis that relies on runtime emulation of programs [19]. `TaintCheck` is implemented as an extension of `Valgrind` [18] and detects vulnerability exploits at the time tainted data is used by the program. `TaintCheck` does not require access or special compilation of application source code. The analyzed program is instrumented at runtime, and the taint analysis code allocates a data structure for each identified tainted data. Each allocated data structure records a copy of the tainted data, a snapshot of the current stack, and a system call number when the tainted data originates from a system call. Each byte of memory (e.g. registers, stack, etc.) has a four-byte shadow memory that stores a pointer to a tainted data structure if that memory location is tainted or a NULL in case of an untainted location. The paper argues that `TaintCheck` is able to detect previously unknown exploit attacks, and automatically create corresponding signatures. `TaintCheck` is meant to be an emulation container for a running application, and thus incurs some performance overhead. The authors of `TaintCheck` reports slowdown of up to 40 times for their benchmark programs. On the other hand, SAINT is meant to detect vulnerabilities during the development phase of an application, does not incur any performance overhead at program runtime.

Libsafe 2.0 by Avaya Labs is a library that detects and avoids some instances of format string vulnerabilities [26]. `Libsafe` operates in three steps: interception, safety check, and violation handling. The installation of `Libsafe` automatically redirects all calls to functions of the C standard library to their equivalent `Libsafe` functions. Once a function call is intercepted, `Libsafe` checks call arguments and either executes the original C function (e.g. `sprintf`), or calls its safer alternative (e.g. `snprintf`). `Libsafe` applies a special handling for functions `IO_vfprintf()` and `IO_vfscanf()` on which all other `*printf` and `*scanf` func-

[5] http://suif.stanford.edu/bddbddb

tions rely. For these two functions, `Libsafe` checks that the associated pointer argument of each %n specifier do not point to a return address or frame pointer. `Libsafe` also checks that all call arguments of the function are contained within a single stack frame. If any of these checks fails, then `Libsafe` found a violation. `Libsafe` handles violation by terminating the running process. There are few exceptions where `Libsafe` may authorize further execution of the process. For a proper behavior, `Libsafe` 2.0 requires special compilation of programs and usage of some specific C libraries. The paper does not report the performance slowdown incur by `Libsafe` usage.

7.3 Hybrid analysis for vulnerability detection

[24]

7.4 Symbolic execution for vulnerability detection

ARDILLA by Kieżun et al. is a tool that uses concolic execution to automatically create input leading to SQL injection and cross-site scripting attacks for web applications written in PHP [10]. `ARDILLA` does not require any annotations. Developers define a time limit within which `ARDILLA` attempts to create input data that exercise security vulnerabilities present in the application. For that, `ARDILLA` uses an input generator to create new input for the application. During application execution, the input generator records path constraints that capture the control-flow paths taken by that specific execution. The input generator then automatically and iteratively creates new input data by negating the obtained path constraints. The web application then runs with the newly created input data and flows along program paths different from previous executions. The input data generation continues until all program paths are covered or the time limit set by the developer exceeds. During program executions triggered by the input generator, `ARDILLA` tracks tainted values. More importantly, `ARDILLA` also tracks tainted values that are stored into the database. That is, tainted values stored into the database are also marked as tainted at their retrieval.

8. Conclusions

In this paper, we have presented SAINT, a whole-program static taint analysis that is flow-sensitive, interprocedural and can be run either with context-sensitivity or without. SAINT computes tainted paths in C programs and is available for download at `https://github.com/xaviernoumbis/saint` SAINT's taint analysis is sound. To the best of our knowledge SAINT is the only tool that implements static taint analysis for a static language using the iterative dataflow framework [11, 17]. The staged nature of SAINT makes it suitable for integration in integrated development environment (e.g: Eclipse, Anjuta, etc.).

A. Complete Algorithm

In this section, algorithm 3 shows the full pseudo-algorithm for the context-sensitive analysis described in Section 5. Only the handling of the `CALL` statement is specific to the context-sensitive algorithm. All other flow functions are common to the intraprocedural and context-insensitive analysis, both described in Section 5 as well.

Acknowledgments

References

[1] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594299. URL `http://doi.acm.org/10.1145/2594291.2594299`

[2] D. Avots, M. Dalton, V. B. Livshits, and M. S. Lam. Improving software security with a c pointer analysis. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 332–341, New York, NY, USA, 2005. ACM. ISBN 1-58113-963-2. doi: 10.1145/1062455.1062520. URL `http://doi.acm.org/10.1145/1062455.1062520`

[3] D. CEARA. Detecting software vulnerabilities static taint analysis. http://tanalysis.googlecode.com/files/DumitruCeara_BSc.pdf, 2009. Internship Report at Verimag Laboratory.

[4] R. Chang, G. Jiang, F. Ivancic, S. Sankaranarayanan, and V. Shmatikov. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, CSF '09, pages 186–199, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3712-2. doi: 10.1109/CSF.2009.13. URL `http://dx.doi.org/10.1109/CSF.2009.13`

[5] C. Cifuentes and B. Scholz. Parfait: designing a scalable bug checker. In *SAW '08: Proceedings of the 2008 workshop on Static analysis*, pages 4–11, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-924-1. doi: http://doi.acm.org/10.1145/1394504.1394505.

[6] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-734-6. doi: 10.1145/1273463.1273490. URL `http://doi.acm.org/10.1145/1273463.1273490`

[7] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 192–203, New York, NY, USA, 1999. ACM. ISBN 1-58113-094-5. doi: 10.1145/301618.301665. URL `http://doi.acm.org/10.1145/301618.301665`

[8] B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 289–298, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-61284-356-8. URL `http://dl.acm.org/citation.cfm?id=2190025.2190075`

[9] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2574-1. doi: 10.1109/SP.2006.29. URL `http://dx.doi.org/10.1109/SP.2006.29`

[10] A. Kieżun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE'09, Proceedings of the 31st International Conference on Software Engineering*, pages 199–209, Vancouver, BC, Canada, May 20–22, 2009.

[11] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM. doi: 10.1145/512927.512945. URL `http://doi.acm.org/10.1145/512927.512945`

[12] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9. URL `http://dl.acm.org/citation.cfm?id=977395.977673`

[13] C. Lattner, A. Lenharth, and V. Adve. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, San Diego, California, June 2007.

[14] O. Lhoták and K.-C. A. Chung. Points-to analysis with efficient strong updates. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 3–16, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926389. URL `http://doi.acm.org/10.1145/1926385.1926389`

[15] V. B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the 14th Usenix Security Symposium*, pages 271–286, Aug. 2005.

[16] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: A program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 365–383, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: 10.1145/1094811.1094840. URL `http://doi.acm.org/10.1145/1094811.1094840`

[17] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. ISBN 1-55860-320-4.

[18] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *In Third Workshop on Runtime Verification (RV03*, 2003.

[19] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2005)*, 2005.

[20] B. Scholz, C. Zhang, and C. Cifuentes. User-input dependence analysis via graph reachability. Technical report, Mountain View, CA, USA, 2008.

[21] B. Scholz, C. Zhang, and C. Cifuentes. User-input dependence analysis via graph reachability. Technical report, Mountain View, CA, USA, 2008.

[22] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*, SSYM'01, pages 16–16, Berkeley, CA, USA, 2001. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1251327.1251343`

[23] M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[24] O. Tripp and O. Weisman. Hybrid analysis for javascript security assessment. New York, NY, USA, 2011. ACM Conference on the Foundations of Software Engineering (ESEC/FSE '11 – Industrial Track).

[25] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: effective taint analysis of web applications. *SIGPLAN Not.*, 44(6): 87–97, June 2009. ISSN 0362-1340. doi: 10.1145/1543135.1542486. URL `http://doi.acm.org/10.1145/1543135.1542486`

[26] T. Tsai and N. Singh. Libsafe 2.0: Detection of format string vulnerability exploits. Technical report, Mountain Ave, NJ, USA, 2001.

[27] J. D. Ullman. *Principles of Database and Knowledge-base Systems, Vol. I.* Computer Science Press, Inc., New York, NY, USA, 1988. ISBN 0-88175-188-X.